VŠB – Technical University of Ostrava
Faculty of Electrical Engineering and Computer Science
Department of Computer Science

Kuchařka pro iPhone
Cookbook iPhone Application

2010 Jiří Urbášek

I declare that I have worked out this diploma thesis by myself. I have stated all the publications and other resources I have used.

Ostrava, 7th May 2010 .....

I would like to thank to Ing. Michal Krumnikl for many inspiring suggestions during the development of my iPhone application, to my parents for preparing very pleasant study background for me and to my girlfriend who helped me to spend wonderful time when I was resting from my school obligations.

Abstrakt

Vývoj aplikací pro iPhone se v současné době stává velmi oblíbeným. V této práci si představíme ukázkouvou aplikaci pro iPhone a nastíníme postupy vedoucí k úspěšnému spuštění aplikace v Apple AppStore. Uvedeme krátký přehled API tříd, které poskytuje Cocoa Touch framework a zmíníme také volně dostupné knihovny rozšiřující toto základní API. Hlavní prinicpy při programování iPhone aplikací se budeme snažit ukázat na krátkých příkladech. Součástí implementace je webová aplikace poskytující RESTful rozhraní, na kterém naznačíme obecná pravidla komunikace iPhone aplikací s internetem.

Klíčová slova: iPhone, Objective-C, Cocoa Touch, REST, kuchařka, webová aplikace, Nette framework, Three20 framework

Abstract

Nowadays, iPhone applications development became very popular. In this document we are going to introduce sample iPhone application and describe approaches needed to successfully launch it in Apple AppStore. We will briefly review iPhone SDK API classes as well as external open source frameworks extending it. We will try to provide short examples to demonstrate basic approaches used while developing iPhone applications. To show principles of communication between iPhone application and rest of the world a RESTful web application is created and general guidelines to access internet on the iPhone will be mentioned.

Keywords: iPhone, Objective-C, Cocoa Touch, REST, cookbook, web application, Nette framework, Three20 framework

List of Acronyms and Symbols Used

API	–	Application Programming Interface
CPU	–	Central Processing Unit
DBMS	–	Database Management System
EDGE	–	Enhanced Data Rates for Global Evolution
ERD	–	Entity Relationship Diagram
GPRS	–	General Packet Radio Service
HTML	–	Hypertext Markup Language
HTTP	–	Hypertext Transfer Protocol
I/O	–	Input/Output
IB	–	Interface Builder
IDE	–	Integrated Development Environment
JSON	–	JavaScript Object Notation
MVC	–	Model View Controller
ORM	–	Object Relational Mapping
PHP	–	PHP: Hypertext Preprocessor (formerly Personal Home Page)
RAM	–	Random Access Memory
REST	–	Representational State Transfer
SDK	–	Software Development Kit
SOAP	–	Simple Object Access Protocol
SQL	–	Structured Query Language
UI	–	User Interface
URL	–	Unified Resource Locator
XML	–	Extensible Markup Language

List of Figures

List of Source Code Examples

Contents

1 Introduction

The purpose of this document is to describe resources and knowledge needed to develop simple application for iPhone or iPod Touch. In the following text we will substitute iPhone or iPod Touch application by word App and all the functionalities described for iPhone application will be valid for application running on iPod Touch as well.

First chapter shortly describes all the hardware and software resources needed to start iPhone applications programming. We will also discuss other things related to iPhone Apps development such as marketing and placing the App into AppStore.

Then we will analyze functionalities required for sample application used as a demonstration of how to use all the things described in this document. We will clarify why did we create two applications and explain development process we used for implementing them.

In the subsequent chapters a basic experience with iPhone SDK and knowledge of Objective-C language is expected from the reader.

The fourth chapter focuses on Cocoa Touch framework for iPhone which is the very basis for application development. We will describe main building stones for Apps such as view-controller concept, table views, core data database layer or unit testing using OCUnit framework.

After a brief introduction to Cocoa Touch we will dive into libraries and frameworks extending standard API. The fifth chapter will cover Three20 framework made by Joe Hewitt, developer of iPhone Facebook App. We will describe several tools which greatly simplify iPhone Apps developing and try to exhibit main functionality of Three20 on short examples.

Afterwards we will explore possibilities of communication between iPhone App and REST services and describe SBJSON library.

At the end we will talk about what we covered and what was excluded from this document and explain why the sample App is not ready to be put in the AppStore yet.

2 Fundamentals of iPhone Development

Nowadays, iPhone application development became very popular (actually, in the United States it has been popular for quite a long time but in the Czech Republic it has started just now, at the beginning of 2010). This platform is very innovative and highly focused on beautiful user interface and great user experience. Apple iPhone or iPod Touch is very interesting for developers because it offers services such as GPS location, accelerometer, Wi-Fi, maps or access to build in camera or recorder.

2.1 Before we Start

iPhone programming is not for everyone. Programmers must meet certain Apple criteria. In fact, the criteria is Apple itself, because it is possible to develop iPhone Apps only on the Mac OS X platform using XCode IDE. Moreover, the only programming language available is Objective-C (there was possibility to develop Apps in .NET or Adobe Flash and *translate* them to Objective-C, however, since iPhone OS 4.0, coming in summer 2010, Apple strictly prohibited this approach).

The Internet is also a necessary tool for iPhone programming. Of course, we do not need internet connection to develop the App itself but it comes very handy for solving problems. An excellent resource is the StackOverflow forum `http://stackoverflow.com/`.

English. There is nothing more to add. Approximately 99% resources are only in English so the developer really needs to be an experienced user of this language.

2.2 During the Development

When we have all the resources needed, we can start developing. Probably we have found some online tutorial and are going to build our Hello World application. There is one more treasure hidden in XCode. We can access it clicking *Help* > *Developer Documentation* or by keyboard shortcut *shift+cmd+?*. The documentation is a really wonderful thing containing hundreds of pages describing API classes and programming techniques including plenty of sample applications. When we are in doubt what some class or method does there is nothing easier than just right click the class or method name and select *Find Text in Documentation.*

XCode, however, is not the only development tool. There are many more tools for building UI, improving application performance, seeking memory leaks or developing iPhone optimized webpages. We will talk about them later.

2.3 When the App is Finished

We need to register with the Apple Developer Program and purchase a one year license to access the AppStore. Without this license we cannot place our Apps in the AppStore and we even cannot run the App on our own device, only in the iPhone Simulator.

Let's suppose we have finished our App. What is the next step? Surely, we want to place it into the AppStore and start earning millions. We can do it immediately, nevertheless it would be the biggest fault we could do and it would probably destroy all the time spent on the development. What should we do then?

We definitely need to create a web page supporting the App, with a simple name and possibly a .com domain. It should describe the reason why the App is the best one and why the customer should buy it. A forum or a blog is very useful too. Great design and user friendliness is a matter of fact.

Viral marketing. Creating a twitter account, creating a facebook fan page, writing an article about the App and encouraging some popular blogs to post it. These things are essentials of starting successful marketing and telling users that our App exists.

2.4 How to Succeed in AppStore

Our App is in the AppStore, we have a nice and outstanding web page, famous blogs are posting about us and our twitter account has a large base of followers. Maybe we are placed in the list of top 25 Apps and in the New and Noteworthy section. We are very lucky if we stay there longer than a few days. It does not end when users find out about our App – that is the beginning.

We should listen to reviews and check the Apps rating. Successful Apps have 4 or 4.5 stars out of a total of 5. People are dynamic and our App has to be dynamic too. When a bug appears it is not possible to fix it after a few months because we are too busy. We should try to review the customer needs and keep adding new features to fulfill them. Apple customers are expecting beautiful design and high level of user experience and we should get used to it as well if we want to build applications for them.

Even if we do all the things described above, there is no guarantee that we will succeed. The risk that someone will come with something similar we have, but better designed and with new magic features, is still here. There is a huge competition in the AppStore and that makes building iPhone Apps even more exciting and challenging.

As an example, around September 2009, there were about 2 Apps similar to our sample Application: Cookbook for iPhone. These days, in April 2010, there are roughly 6 or 7 of them, so the market share for this field has been mostly covered.

3 Sample Application Analysis

As a sample application we have chosen application for managing recipes database and called it Cookbook for iPhone. Because we want to demonstrate communication between iPhone and world wide web a web application have been also implemented.

3.1 iPhone and Web Application

The basis for all the data is multiuser web based application. It maintains database of recipes and has following functionality:

- managing user accounts
- user can create/modify/delete recipes and upload photos
- user can comment on recipes
- recipes are categorized into categories according to ingredients
- RESTful interface to manage the data remotely is provided

The other part is an iPhone client application which can connect to the web application. Its users have functionalities such as:

- browse content of the web application, search for recipes containing certain ingredients
- store recipes of interest into local database
- generate grocery list from recipe ingredients
- simple unit conversion and timer

3.2 Data Structure

Data store plays main role in our concept. Because the main principle – managing database of recipes – is the same in both applications we do not need to create two distinct database schemas. The general schema is shown in figure 1.

3.3 Applications Development Workflow

Several technologies have been used to develop desired applications.

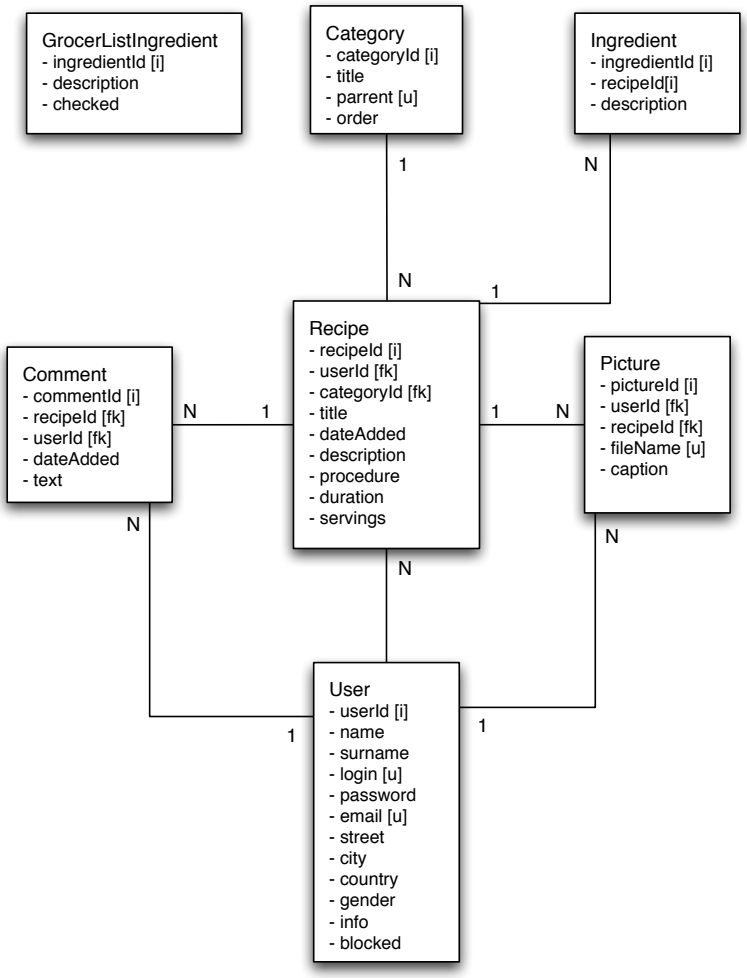

Figure 1: Database ERD diagram

3.3.1 Nette PHP Framework

Web based application is built on top of Nette PHP framework. Nette framework simplifies web development and offers three layer architecture known as MVC. It has one of the most growing community in the Czech Republic which helps to solve problems very quickly.

Database background is supported by dibi PHP library acting as „tiny 'n' smart database layer". This library allows to abstract database connection so that we can use different kinds of DBMS systems without any change in source codes.

3.3.2 iPhone SDK

iPhone SDK (Software Development Kit) is a set of tools for developing iPhone applications. It can be downloaded from Apples iPhone Dev Center (`http://developer.apple.com/iphone/`) for free and consists of XCode IDE, Interface Builder (IB) for creating GUI, iPhone Simulator for running applications and many other tools.

Application Programming Interface (API) is supplied by Cocoa Touch a set of frameworks used to control iPhone device. The most used ones are Foundation framework accomplishing basic tasks such as working with strings, arrays, exceptions or networking and UIKit framework containing classes for creating user interface. Other frameworks have been also used such as CoreData for controlling application data, CoreGraphic for animations, AudioToolbox for playing sounds or SenTestingKit to support unit testing in Objective-C.

iPhone client application is build on top of Three20 framework which is open source framework extending standard UIKit and Foundation frameworks. We will focus on Three20 later in this document.

3.3.3 Development Process

The **waterfall model** is common approach for developing various applications. Its principle is to go sequentially from requirements specification, designing and implementation of whole the application to verification and testing phase. After the testing phase application is ready to be shipped to the customer. During the sample application development we tried to avoid this principle because it does not fit todays needs.

Agile development methods are becoming more and more popular because they are flexible enough to be able to respond customer needs and change application functionalities in the middle of the development process. General principle of agile development methods is to build and test only a little piece of application functionality and deliver it to the customer. If the customer is satisfied another piece of functionality is developed. The methods do not require huge documentation and software analyses. Rather they start with high level application structure and each functionality is specified in more detail in the time it is developed.

4 Cocoa Touch Overview

4.1 Cocoa Touch Patterns

There are some common design patterns widely used in iPhone programming. Some of them are similar to those well known from GoF book[1] and some of them are more or less unique. Of course, there are many patterns implemented in Cocoa Touch framework. We will not cover all of them, rather we will focus on the most important and most used ones.

4.1.1 Model-View-Controller (MVC)

A lot had been said and written about the definition of MVC pattern, we will focus on its actual use in the Cocoa Touch framework instead.

To *find* MVC in Cocoa Touch is not too difficult. There are the `UIView` and `UIViewController` classes. However, where is the model part? Prior to iPhone OS 3.0 no model part had been defined. Programmers had to build their own set of classes to store data into files or SQLite database. Nowadays, the situation has changed because the CoreData framework has been ported from Mac OS to iPhone OS, thus offering developers a fast and easy to use database layer. Views, controllers and core data will be covered later as separate parts, but the question is: How do they communicate together? A delegation pattern is frequently used for this purpose.

4.1.2 Notifying Delegates

A delegation pattern can be thought of as something between Observer pattern and Strategy pattern from GoF. It is based on protocols.

The idea is that an object is interested in the others behavior. The object of interest needs to have a delegate property adopting delegates protocol. Every time the object of interest does a certain task, it calls a particular method on its delegate. The delegates methods can have purely informational meaning or can completely move a certain task to the delegating object. It is also very common to pass the object of interest as an argument to its delegate methods. We will discuss the `UITableViewDelegate` as an example.

There are methods informing that something has happened, such as:

```
tableView: willDisplayCell :forRowAtIndexPath:
tableView:willSelectRowAtIndexPath:
tableView:didSelectRowAtIndexPath:
tableView:willBeginEditingRowAtIndexPath:
```

In these methods we can influence the behavior of the table view. For instance, we can respond to user taps, because we get notified every time a user selects a row in a table (`tableView:didSelectRowAtIndexPath:`).

On the other side, there are methods that can directly alter a table view appearance:

[1] Design Patterns: Elements of Reusable Object-Oriented Software

```
tableView:heightForRowAtIndexPath:
tableView:shouldIndentWhileEditingRowAtIndexPath:
tableView:titleForDeleteConfirmationButtonForRowAtIndexPath:
```

Using these methods we can directly affect the process of displaying of the table view. We can change the height for any table row
(`tableView:heightForRowAtIndexPath:`).

To use delegates we need to have a delegate protocol defined and our object of interest has to have a property conforming to this protocol. However, there is also an easier way how to respond to certain actions.

4.1.3 Target-Action Mechanism

This mechanism enables a control object such as a button, text field or a slider to send a message to another object when something specific occurs. For instance we can define that when a button is taped, a certain method will be called on a certain object with the button as its argument.

Objects which can perform such actions are those subclassing the `UIControl` class and some others such as bar button item. There is the `addTarget:action:`
`forControlEvents:` method defining that when a certain event is fired
(*ControlEvents*), a method (*action*) will be called on certain object (*target*). Typical usage can be handling of `UIButton` click represented by event
`UIControlEventTouchUpInside`:

```
UIButton *button = [[UIButton alloc] init];
[button addTarget:self
           action:@selector(handleButtonClick:)
 forControlEvents:UIControlEventTouchUpInside];
```

Exhibit 1: Example of handling click event on button

There are 18 other events to handle, as well.

The action method can have three forms depending on the amount of information we need to pass to it.

```
- (void)action
- (void)action:(id)sender
- (void)action:(id)sender forEvent:(UIEvent *)event
```

Exhibit 2: Action method variants

The *cleverest* one takes the object which will fire the event (e.g. button or slider) and the `UIEvent` object representing the fired event as its arguments.

4.1.4 Key-Value Observing (KVO)

This principle allows to get notified when a property of any object changes its value. Apple documentation says the following:

„In the context of the Model-View-Controller pattern, key-value observing is especially important because it enables view objects to observe – via the controller layer – changes in model objects“

The documentation also provides an example:

```
// register "inspector" to receive change notifications
// for the "openingBalance" property of the "account" object
// and that both the old and new values of "openingBalance"
// should be provided to the observer
[account addObserver:inspector
          forKeyPath:@"openingBalance"
             options:(NSKeyValueObservingOptionNew | NSKeyValueObservingOptionOld)
             context:NULL];
```

Exhibit 3: Example of key-value observing

A class which wants to be successfully observed has to be Key-Value Coding (KVC) Compliant. It means it has to follow rules of KVC Compliance for property accessors. These rules are described in the Apple documentation on page Ensuring KVC Compliance.

4.2 Displaying Application Content

As had previously been said, Cocoa Touch framework is based on MVC design pattern. Views are represented by subclasses of `UIView`, controllers are subclasses of `UIViewController` and a model can be represented using CoreData framework which we will cover later in this document.

4.2.1 Using Views and Windows

Content of an App can be displayed either in a window or in views. There can be only one window in the whole App so it is a common approach to add multiple views into the application window and switch them as needed.

`UIView` class is a superclass of all views and it does not provide much functionality itself. It is meant to be subclassed. There are some `UIView` subclasses already prepared in UIKit. For instance `UIWindow` already mentioned, `UIScrollView` for displaying content bigger than screen size, `UITableView` for displaying a list of values, `UIImageView` for displaying one or more images, `UIAlertView` for showing warning and error messages or `UIControl` subclasses such as `UIButton` or `UISlider`.

Moreover, `UIView` is a subclass of `UIResponder` which means that views can respond to touches. In an iPhone App it is possible to handle only five distinct touches (e.g. five finger taps) at the same time.

Views are designed as composite objects. So they are able to hold other views in themselves. And we should really add subviews to our custom views.

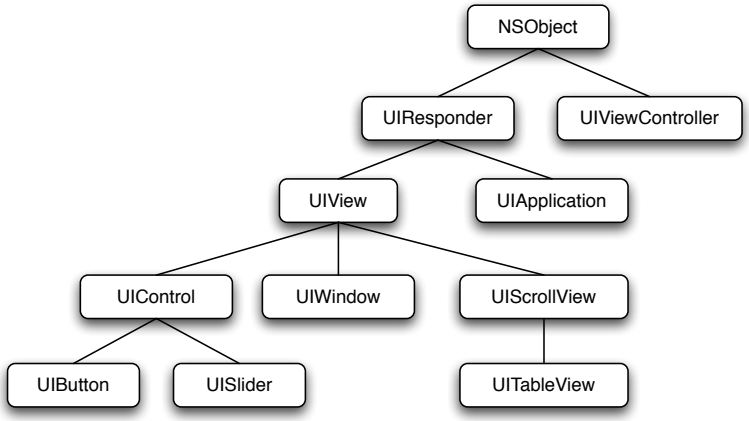

Figure 2: Hierarchy of UIKit classes related to views and controllers

4.2.2 Creating Custom Views

We can create a custom view either programmatically or using the Interface Builder (IB), a part of Developer tools. However, when should we use the IB and when is it better to create the view by hand?

We should create custom views programmatically if they are going to be either very complex – full of subviews, or very easy. It may sound as a contradiction but sometimes e.g. in case of `UITableView` we do not need any extra functionality over a standard UIKit class so it would be wasting of our efforts, as well as wasting of time in runtime, to use the IB for creating such an easy view. On the other side, the IB has limited functionality, therefore views with complex functionality and behavior are also better to be done by hand, because we have full control over them.

For other cases we can use the IB without any worries. The IB will store the defined view as an `.xib` file in our application bundle. Xib files are in fact XML files (xib file means xml representation of `.nib` file) defining the properties of the view, its subviews and other information. Despite of their extension, xib files are often referred as nib files used in the Cocoa framework to develop Mac applications.

Here are a few methods we will want to know about when creating views programmatically:

- `initWithFrame:` creates a new view *living* in a specified frame. Each view has to have its frame, which specifies its position relative to its superviews frame.

- `autoresizeSubviews`, `autoresizingMask` properties specify whether subviews can be resized and what is the type of the autoresizing policy. These properties are useful especially when we want to allow our App to handle different interface orientations properly.

- `addSubview:` adds a subview into view hierarchy. The child view is displayed above its underlying view.
- `backgroundColor` property does exactly what its name suggests – it sets the color of the background of the view.

4.2.3 Controlling Views

As a controller there is the `UIViewController` class and just like `UIView` it has some subclasses in UIKit such as `UITableViewController`, `UITabBarController` or `UINavigationController`. Again, there is no point in using `UIViewController` as it is. It needs to be subclassed.

`UINavigationController` is a special type of a view controller. It has a build-in navigation bar which allows to navigate within the hierarchy of view controllers very easily. By default, it provides a button for navigating on the left side of the navigation bar, a title in the middle and on the right side it can optionally display a custom button. Frequently used is the `editButtonItem` button which adds possibility to edit table view items almost without any effort.

When we are subclassing `UIViewController`, there are several methods we should know about:

- `init`, `initWithNibName:bundle:` methods will initialize the view controller. The latter one we should use when we have created our view using IB to specify its nib name and bundle where the views nib is stored
- `loadView` method initializes the `view` property of the view controller. The default implementation searches for the nib file and creates a view using it. However, if we are creating the `view` programmatically we can do it in this method, but must not call `[super loadView]`
- `viewDidLoad` method is called automatically when a view has been loaded using the `loadView` method. Here we typically perform additional initialization steps when the view has been loaded from nib
- `viewDidUnload` is called when the view controller needs to release the view. Here we typically release all the `IBOutlet`s created in the IB
- `viewWillAppear:` notifies view controller when the view is about to appear. If we have a tab bar application, this method will be called every time the tab bar item is tapped. That is the contrast to `viewDidLoad` method called only when a view is displayed for the first time. We can perform here operations such as table view data reloading to ensure that the user has always correct data displayed.
- `didReceiveMemoryWarning` says that the view controller consumes to much memory. This method is a step before quitting our App by system so we should consider to free all the objects and other resources stored in memory which are not needed at a time and recreate them later.

- `shouldAutorotateToInterfaceOrientation:` method asks whether the view layout can be changed when the interface orientation changes. This happens when the user rotates his/her device.
- `presentModalViewController:animated:`, `dismissModalViewControllerAnimated:` shows and dismisses another view controller. This controller is not inserted into the navigation stack, it is displayed using special animation instead.
- `navigationItem` property gives an access to the navigation bar of the underlying navigation controller to alter the navigation behavior. We should not use this property if we are not using a navigation controller.
- `navigationController` property gets the underlying navigation controller. If the view controller is not in the navigation stack, it will return `nil`. The typical usage is to get the navigation controller and push new view controller into navigation stack when table view item is tapped.
- `tabBarItem` property specifies an item in the tab bar, its title and image, for a particular view controller. If we are not using a tab bar in our App we should not access this property.

4.3 Table Views

Table view is one of the basic tools for displaying application content. It symbolizes a list of some values. The table view advantage is that it can be customized to display almost anything.

To learn how the table view works we need to understand its five basic parts:

- `UITableViewController`
- `UITableView`
- `UITableViewCell`
- `UITableViewDataSource`
- `UITableViewDelegate`

We have already discussed the MVC patten, therefore it is obvious that the `UITableViewController` is responsible for managing `UITableView`. The purpose of the `UITableViewDelegate` is to inform about the behavior of the table view. However, there is also the `UITableViewDataSource`. It is basically a kind of delegate as well, a delegate with specific functionality. It handles access to data displayed in the table view. Very roughly speaking it can be considered as a kind of model for the table view. Because the table view represents a list of values, for each such value it is convenient to have its own view. This is the purpose of `UITableViewCell`. It is a `UIView` subclass so it can use all advantages that the `UIView` has.

Table view consists of one or more sections. Each section is a group of related items referred as rows and can have its own header and footer. The position of a cell in the table view is defined by index path specifying in which section on which row a table cell is placed.

4.3.1 Displaying Data in a Table

In simple tables, a controller can also act as a `dataSource` and a `delegate`. The data source and delegate play the main role when defining content of a table view. There are three methods we will definitely need to implement to accomplish the most basic behavior:

1. `tableView:cellForRowAtIndexPath:` is responsible for defining the cell to be displayed at specific index path
2. `tableView:numberOfRowsInSection:` tells how many items/cells will be displayed in specific table section
3. `tableView:didSelectRowAtIndexPath:` handles users tap to certain table row. This is the place where we navigate to another view controller or place checkmark on the table row.

We assign data source and delegate objects to the controller properties either in the IB, by dragging `dataSource` and `delegate` outlets to File's Owner, or programmatically by assigning them in the `init` method or `viewDidLoad` method.

Due to the fact that the table view can contain lots of items, table view cells are implemented as reusable objects. It means that instead of creating new cells when scrolling table view, those which are not currently visible will be reused and only their properties will be changed. This significantly lowers the memory usage as well as the number of created objects.

4.3.2 Customizing of a Table View

Table views are highly customizable. The table view cells are mostly subjects of change. The UIKit provides four basic table view cell styles. The style of the cell has to be specified when initializing it by method `initWithStyle:reuseIdentifier:`. Unfortunately the style names are ending such as `Default`,`Value1`, `Value2` and `Subtitle` which does not say much about how does it look like.

The text font, size or color of the cell we can change by using the `textLabel` property. We can also add an image into a cell by defining the `imageView` property.

We can add a checkmark character to a cell by specifying the `accessoryType` property to value of `UITableViewCellAccessoryCheckmark` but there are also types of disclosure indicator or disclosure button.

It is possible to change only appearance of a selected cell by using

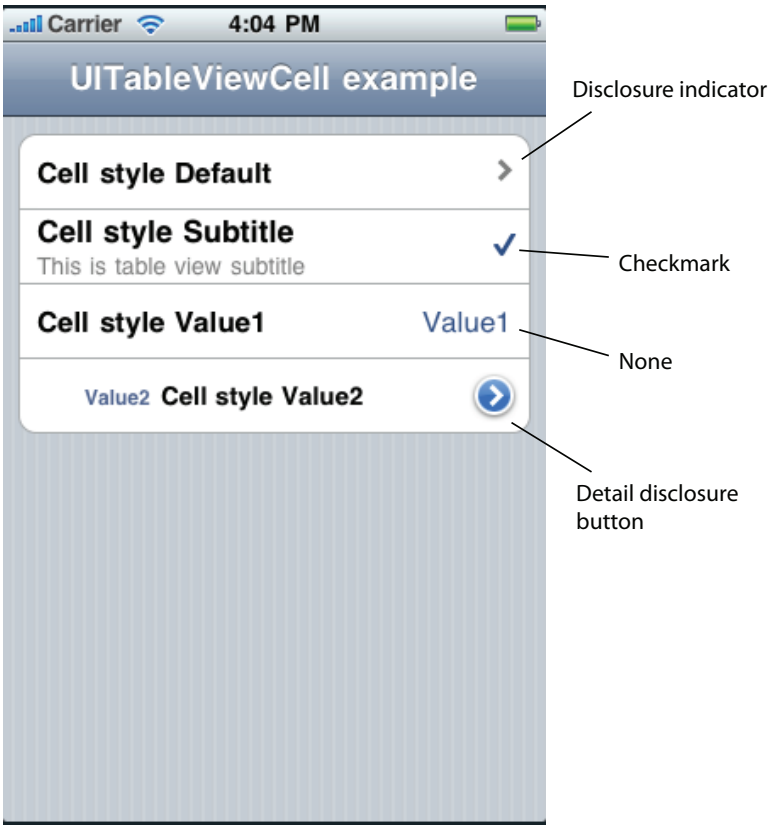

Figure 3: Table view cells example

`selectedTextColor` property or specify the `highlightedImage` property of the `imageView` cell property.

Because `UITableViewCell` is a subclass of `UIView`, we can easily add new subviews to existing cells. For instance, we want to add a slider, text field or a *YES/NO switch* into a table row. It is very easy. We just create a new instance of the desired control and add it as a subview of the table view cell.

It is also very common to create custom table cells by subclassing `UITableViewCell` and defining its content as in any other view. However, when subclassing a table view cell, we have to keep in mind several things: firstly, to implement properly reusing of the cell in `init` method and secondly, issue `[self setNeedsDisplay]` every time any property displayed in the cell changes its content.

4.4 Form Components

It happens very often that we need to handle some user action in our App. We can ask the user to allow or disallow some functionality, to pick value from range of values, to enter some text or to perform an action on button tap. UIKit has a number of controls designed to simplify such tasks. All of them are subclasses of `UIControl`, they are thus able to perform actions on defined events. Moreover `UIControl` subclasses `UIView` so all the controls can have subviews and other features similar to views.

4.4.1 Read-only Content and Actions

Sometimes we only need to show information which will never change such as titles etc. For that reason **`UILabel`** is suitable. It can be customized to display different text font, text size, text color, to be left or right aligned, to contain one or more lines of text and a lot of more. It is not a `UIControl` subclass so it cannot respond to system events.

On the other side there is the **`UIButton`** which is meant to perform an action on a tap. We typically register it to listen for `UIControlEventTouchUpInside` event as described by the Target-Action principle in section 4.1.3 and in action method called we perform desired task. If we do not want a button to listen for events we can set `enabled` property to `NO` and the button will no longer be able to receive taps on it. We can also highlight or select a button manually using `highlighted` and `selected` properties. It is very common to specify a button background image which we can accomplish due to the fact it inherits from `UIView`.

4.4.2 Picking a Value

There are several ways how to let user pick a value. It depends on whether we need to get a boolean value or whether we select a value from a set of predefined values. The easiest kind of interaction is to ask the user *yes or no?* using switch control.

`UISwitch` looks like an on-off button used in Preferences App. It has a very simple interface to communicate with - just one method and one property. The `on` property says

whether the current value is on or off and method `setOn:animated:` put value to on or off depending the boolean value given in the first argument.

`UISegmentControl` can be described as a bar containing several buttons, each button is called segment. To define the segment appearance we use method `insertSegmentWithTitle:atIndex:animated:` or method `insertSegmentWithImage:atIndex:animated:` eventually. To find out which segment has been tapped we register the action method for `UIControlEventValueChanged` event. In this method we ask the object given in argument – an instance of `UISegmentControl` – for its `selectedSegmentIndex` property, which will return the index of the selected segment.

Very original is **`UIDatePicker`** control using `UIDatePickerView` for displaying multiple rotating wheels to specify date or time. There are several modes for date picker defined by `UIDatePickerMode` enumeration:

- time mode displays hours and minutes
- date mode displays days months and years
- date and time mode contains both of former ones
- count down timer displays hours and minutes but disallows to select time 0:00.

To get the value selected by the picker we register it to listen for event `UIControlEventValueChanged` and in the action method we get the value as `NSDate` object using `date` property.

4.4.3 Getting Text Input

Getting text input can be quite complicated task because we have to manage keyboard to appear when a text input field is tapped, we can choose several different keyboard layouts to work with and when editing is finished we want the keyboard to disappear.

This complexity is not case of **`UITextField`** class. This class is very simple to use and the system will manage a lot of things for us. To be honest, however, it can be tricky sometimes to accomplish the desired behavior.

The text field looks like a rounded rectangle where, while tapping inside, a blinking cursor will appear and a keyboard will automatically pop up. If we do not like the rounded rectangle border we can choose another one using `borderStyle` property. By default the text field is empty but we can put the text into it manually using the `text` property or we can display a placeholder text which will disappear when the text field is tapped. The keyboard is managed for us, but we can influence its type – define whether the field will serve number pad keyboard, phone pad keyboard, keyboard optimized for entering email address or another one from the types available. All the keyboard behavior is defined by `UITextInputTraits` protocol that `UITextField` class adopts.

The tricky thing when using text fields is making the keyboard to disappear, referred as dismissing, because it does not happen automatically. The `UITextField` class has

a property conforming to `UITextFieldDelegate` protocol which handles this. The protocol among others declares `textFieldShouldReturn:` method performed every time *Return* or *Done* key on the keyboard has been tapped. In this method we call `resignFirstResponder` on text input object given in argument and the keyboard will disappear. A problem can arise when we are using keyboards without *Return* or *Done* key such as num pad. In this case we usually insert a tab bar button into the navigation bar and the user is supposed to tap this button when finished.

`UITextField` is intended only to insert single line text. The *Return* key tap will end up editing and dismiss the keyboard. In some cases, however, we need to insert longer multi lined text. **`UITextView`** will help us with that. It looks like a normal view with no borders because it is intended to fill all the screen width. When we tap on it, the keyboard will appear and we can start editing. If the entered text is too long that it goes under the keyboard, the text view will automatically scroll down to make the text visible. The *Return* key of the keyboard does not dismiss it so the only way how to do that is to add an extra button into the navigation bar as described for text field.

4.5 CoreData - Smart Data Layer

Since iPhone OS 3, CoreData framework has been implemented for this platform too. From that time onwards, programmers can easily build well-factored Apps based on MVC model.

CoreData framework is not ORM, but it is very similar. The definition from Apple says:

„Core Data is a schema-driven object graph management and persistence framework."

Its main aim is to easily store and restore objects and relationships between them in a disk storage. It includes tools for maintaining all the changes to data storage, offers automatic undo/redo support, it is easy to integrate into UIKit table view classes and has infrastructure for versioning data store and migration to newer one.

Because memory management on the iPhone is crucial, it has been designed to store in memory only objects necessarily needed to be there. For this purpose a system of faulting and uniquing of the objects is there.

- Fault objects are objects which have not been fully loaded into memory. These object are automatically loaded when they are asked for. This is very common for loading objects represented by unique keys in relational databases.
- Uniquing is principle which ensures that there are no two same objects loaded in the memory at the time.

CoreData uses a schema to describe the structure of a model objects. This schema is defined in `.xcdatamodel` file and is very similar to an ERD diagram. XCode offers ability to create this schema, manage object properties, as well, as export model objects as custom classes allowing us to implement additional functionality to them.

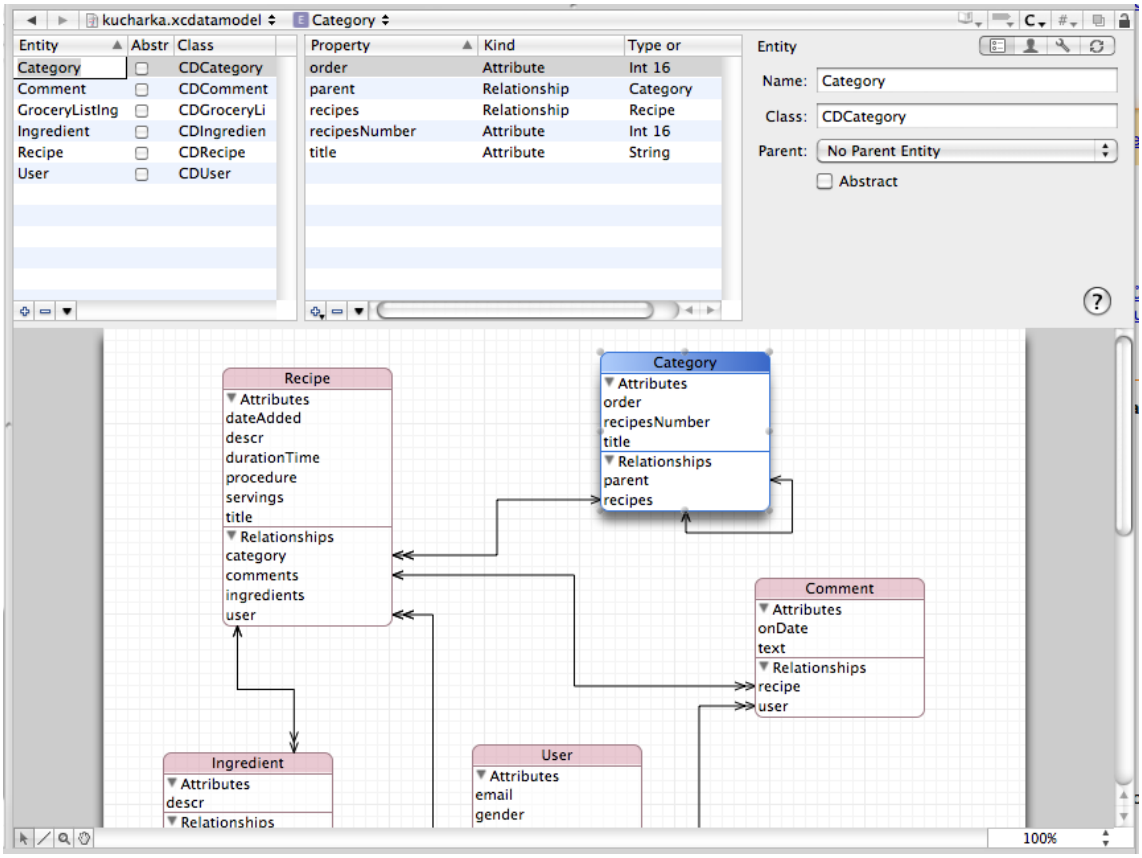

Figure 4: .xcdatamodel file editor interface

4.5.1 Data Storage Management Classes

When manipulating the data store we need to be familiar with several classes:

- `NSManagedObjectModel` defines what we are storing in the data storage. It describes structure of entities, their attributes, relationships between them. ERD like diagram created in `.xcdatamodel` file is basically a graphical representation of this object.
- `NSPersistentStoreCoordinator` handles storing of raw data. This object knows how the data are physically represented in storage file, whether they are stored in SQLite database, as XML file or binary file and how to access them. Note that CoreData for iPhone can only store data in SQLite database.
- `NSManagedObject` represents data of one record for certain entity. In relational database it would be one row in the entity table.
- `NSManagedObjectContex` behaves as a mediator between persistent store which is completely transparent to a programmer and managed objects which the programmer uses the most. This object manages storing and retrieving of actual data, undo/redo functionality as well as it handles proper memory management for data loaded in memory.
- `NSEntityDescription` provides information about concrete entity type. Information about its name, attributes and relationships. This class is also responsible for inserting new records (managed objects) into data store.
- `NSFetchedResultsController` is intended for fetching data from storage efficiently and provides interface suitable for displaying data in table views. It offers fetching based on attributes conditions, result sorting, caching fetched data or managing its batch size.

4.5.2 Accessing Data Store Within App

When creating new iPhone App project in XCode we can check the option *Use Core Data for Storage*. This will generate a set of properties and methods in application delegate for managing CoreData storage. These methods and properties are:

```
@property (nonatomic, retain, readonly) NSManagedObjectModel *managedObjectModel;
@property (nonatomic, retain, readonly) NSManagedObjectContext *managedObjectContext;
@property (nonatomic, retain, readonly) NSPersistentStoreCoordinator *persistentStoreCoordinator
    ;

- (NSString *)applicationDocumentsDirectory;
```

Exhibit 4: Core data stack classes

They will automatically create data store according to the schema defined in `.xcdatamodel` file and allow us to easily add, remove or edit new data.

4.5.3 Manipulating Data Store Records

To add a new record we need to call convenient method `insertNewObjectForEntityForName:inManagedObjectContext:` on `NSEntityDescription` class. We supply the entity name and managed object context instance and as a return value we get a new managed object which we can modify as we need.

To delete data we need to have an existing managed object. Then we issue `deleteObject:` method on instance of managed object context.

Editing of records is also very simple. `NSManagedObject` has method `setValue:forKey:` where we can supply new value and name of an attribute to be changed.

But all these changes are made only in memory. To permanently store them into the file storage we have to save the state of managed object context. For this purpose there is method `save:` which takes a `NSError` instance as its argument and returns the `BOOL` value to indicate the success of the saving process.

4.5.4 Using Custom Model Objects

`NSManagedObject` class is quite simple in therms of data accessing. Usually, it is convenient to extend this class and create custom model objects. XCode is very friendly with this. It can generate those classes for us without any effort. The procedure is following:

1. we specify a custom class name instead of `NSManagedObject` for entity in *Entity* window in `.xcdatamodel` editor
2. we make sure the `.xcdatamodel` file is edited in the internal editor and right-click anywhere in *Group & Files* window where we want the classes to be generated and choose *Add > New File...*
3. a wizard to add new file appears. In the section *Cocoa Touch Class* there is a new item called *Managed Object Class*. We select it and click *Next*
4. next window will ask for destination of newly created classes. We are fine with that so we click *Next*, then we check all the classes we want to generate and click *Finish*.
5. now we can see that `NSManagedObject` subclasses have been generated with properties similar to entity attributes. Note that properties are synthesized as `dynamic`, it means that core data framework will manage access to them dynamically by itself.

4.5.5 Providing Default Storage

What can be confusing or sometimes annoying is that XCode does not offer any built-in possibility to provide default data storage for an App. Let us suppose we have a flight searching App, it is very essential to provide a list of airports and flight companies locally not fetch it from the internet every time the App starts because it changes minimally.

To provide default data store for an App we have to insert all the data manually, save the managed object contexts state and copy `.sqlite` file it will generate. Then we put this file as a resource in our application bundle and modify the `persistentStoreCoordinator` accessor method in AppDelegate (or anywhere it is defined) to use the sql file from application bundle as an initial data storage.

This approach will ensure that when user installs the App on his/her device, the data store will be initialized by the data from the sql file in the application bundle.

4.6 Unit Testing in Objective-C

To support unit testing in Objective-C, there is OCUnit framework developed by Sen:te company(`http://www.sente.ch`). This framework includes the SenTestingKit framework and set of utilities and makefiles to integrate unit tests into XCode project.

4.6.1 2 Types of Tests

In XCode we can create two types of tests. Logic tests and Application tests.

1. Logic tests are unit tests for testing code in „clean-room environment“. It means that each unit test runs independently to the others. We typically test class methods or collections of classes to ensure everything behaves correctly. These tests are easy to set up and run.

2. On the opposite side, application tests are aimed to test a running application. We can run these tests only on a physical device. They can test user interface controls behavior or the device hardware such as GPS location services or accelerometer.

In this document we will cover only logic tests.

4.6.2 Testing Target

Logic tests require their own application target. We create it by clicking *Project* > *New Target...* and selecting *Unit Test Bundle*. Then we specify the target name, for example `Logic Tests`. After that we create a new group in *Group & Files* window called *Tests* which will contain all the testing classes.

The testing class is a class extending `SenTestCase` class. We create it by right clicking *Tests* group, selecting *Add* > *New File...* and in the wizard window we select *Objective-C test case class*. In the next step we type the class name and make sure only the *Logic Tests* target is checked. A test class with a default template will be generated.

When we want to run test cases we change the active target to *Logic Tests* target and build the project normally. Now when there are any fails in the tests we should see them as compile errors in the text editor.

4.6.3 Writing Test Cases

We need to import the class we want to test in a test case file and drag its `.m` file into the *Compile Sources* folder under *Logic Tests* target.

To perform initial definitions before test case starts and after it finishes we can use methods `setUp` and `tearDown`. In the former one we initialize the tested object or perform any other necessary initializations and in the latter one we release everything we had initialized in the `setUp` method.

The test case method takes no arguments, returns void value and its name has to begin with the `test` word. It has following structure:

```
- (void) test<test_case_name> {
    ...      // Set up, call test-case subject API.
    STA...   // Report pass/fail to testing framework.
    ...      // Tear down.
}
```

Exhibit 5: Test case method structure

Sen testing kit provides plenty of assertion macros, we will explore the most common ones:

1. `STAssertEqualObjects` takes three arguments, the first two – expected result object and actual result object – are compulsory, the third one is a formatted string specifying error message used when the test fails. Equality of tested objects is determined by `isEqualTo:` method.

2. `STAssertTrue/STAssertFalse` macros check whether the given expression is true or false respectively. They also take an optional formatted string as a custom error message in the last argument.

3. `STAssertNil/STAssertNotNil` check its first argument for `nil` value. An optional error message is present here as well.

4. `STAssertThrows` expects that expression given in the first argument raises an exception. If not, the test will fail. Again, the last argument specifies a custom error message string.

Complete list of all macros available is on page Unit-Test Result Macro Reference in Apple documentation

4.7 Additional Developer Tools

4.7.1 Interface Builder

Interface Builder is tool to design and test user interface. It allows to drag and drop UI components into windows and views, register outlets as class properties and assign action methods to default system events such as Touch Up Inside event. It is possible

to run created user interface in iPhone Simulator without even having XCode project containing it created. All the Interface Builder files are saved as `.xib` files, views and components are referred as outlets.

There are several objects in xib file:

- *File's Owner* represents controller class owning the xib file. When we need to assign certain outlet to a class property or an action method called when any event fires we drag the outlet or the event to the File's Owner and list of available properties or methods will be displayed automatically.
- *First Responder* represents object responding to user interaction. In the xib file we usually do not alter anything about this object.
- Custom outlets such as views, buttons and other components. These components we usually assign to File's Owner properties or define action methods called when certain event fires.

4.7.2 Instruments

Instruments is very powerful tool to improve application performance. It can track information about object allocations, memory leaks, threads, CPU usage or I/O operations.

In case of memory leaks we can examine all the classes and their memory usage. Instruments automatically detects memory leak and track its stack trace. Thus we can effectively find method where the leak happened and repair our code to work correctly.

To test our application we select *Run* > *Run with Performance Tool* and chose *Object Allocations*, *Leaks*, *CPU Sampler* or any other tool we want the App to be tested by.

4.7.3 Shark

Shark performance tool is intended to analyze application performance and the time it spends for certain operation. We can test entire system or any specific running process.

To test our application we run Shark and choose *Process* value from drop down menu. Another menu listing all available processes to track will appear. From there we select *iPhone Simulator* and click *Start* to begin analyzing it. After that we can run our iPhone application in the simulator and perform operations we consider to be time consuming.

Apple documentation explains how Shark works:

„By default, Shark creates a profile of execution behavior by periodically interrupting each processor in the system and sampling the currently running process, thread, and instruction address as well as the function callstack."

Information gathered by Shark can be also examined by other performance tools to provide more details about what is happening inside the application.

5 Three20 – Third Party iPhone Framework

Three20 is very powerful framework which greatly simplifies iPhone Apps development as well as provides some extended functionalities to standard UIKit behavior.

In this chapter we are going to explain some features used in the sample App.

5.1 Navigating Between View Controllers

Three20 framework offers powerful tools to simplify navigating between view controllers. The tools are the `TTNavigator` class plus `TTURLMap` class. Those classes allow us to use our iPhone application almost like a web application - to show view controllers based on URL which user of the App visits.

This approach tries to simplify and standardize way of navigating within the App.

5.1.1 Basic TTNavigator Setup

Each App using `TTNavigator` has to have more or less same startup initialization. It involves:

- instantiating of `TTNavigator` class, and setting its properties
- setting up a map of URLs and corresponding view controllers in object of `TTURLMap`
- handling persistence and setting startup URL
- defining way of handling incoming URL requests

All of this is typically done in `applicationDidFinishLaunching` delegate method. Most common instantiation code looks as follows:

```
- (void)applicationDidFinishLaunching:(UIApplication *)application {

    TTNavigator* navigator = [TTNavigator navigator];
    navigator.window = [[[UIWindow alloc] initWithFrame:TTScreenBounds()] autorelease];

    TTURLMap* map = navigator.URLMap;

    [map from:@"*" toViewController:[TTWebController class]];

    if (![navigator restoreViewControllers]) {
        [navigator openURLAction:[TTURLAction actionWithURLPath:@"tt://tabBar"]];
    }
}

- (BOOL)application:(UIApplication*)application handleOpenURL:(NSURL*)URL {
    [[TTNavigator navigator] openURLAction:[TTURLAction actionWithURLPath:URL.
        absoluteString]];
    return YES;
}
```

Exhibit 6: TTNavigator instantiation

The main idea of `TTNavigator` is that any action such as a button click or a table view cell tap leading to push another view controller is supposed to open an URL string. These URLs are mapped into a `TTURLMap`.

When instantiating the `TTNavigator` we can also specify the level of persistence by setting `persistenceMode` property or enable the feature of reloading `TTModels` by shake gesture by setting property `supportsShakeToReload` to `YES`.

5.1.2 Choosing Navigation Behavior

The most interesting and most difficult when defining navigation behavior is creating maps for `TTURLMap`.

`TTURLMap` offers plenty of methods for defining almost any type of navigation we need.

5.1.3 TabBar Using TTNavigator

To enable tab bar in our App we need to create a mapping from URL to shared view controller. Then we subclass `UITabBarController` and specify URLs for tabs in `viewDidLoad` using category method `setTabURLs:`.

```
// in applicationDidFinishLaunching method
[map from:@"tt://tabBar" toSharedViewController:[TabBarController class]];

// subclassing UITabBarController class
@interface TabBarController : UITabBarController {
}
@end

@implementation TabBarController

- (void)viewDidLoad {
  [self setTabURLs:[NSArray arrayWithObjects:@"tt://category/",
                                             @"tt :// search",
                                             @"tt :// grocery",
                                             @"tt :// archive",
                                             @"tt :// extras",
                                              nil ]];
}

@end
```

Exhibit 7: Setting up tab bar controller in TTNavigator

Tab bar icon and title we define in each view controller for the particular tab bar item.

5.1.4 Passing Data to Pushed View Controllers

Usually we need to pass some data to view controllers pushed into navigation stack. We can use several approaches to accomplish that.

- no data passed – when specifying map such as:

```
[map from:@"tt://search" toViewController:[SearchFormViewController class]]
```

  A new view controller is pushed into the stack by allocating it and calling `init` method on it.

- passing string value – we can specify map as:

```
[map from:@"tt://category/(initWithCategoryId:)" toViewController:[
    CategoryListViewController class]]
```

  In this case a view controller is instantiated by method `initWithCategoryId:`. Now when we open URL action with URL `@"tt://category/1"` string `@"1"` will be passed as the argument for `initWithCategoryId:` method.

- passing *big data* – when we want to pass data larger than just a simple string we may redefine method `didSelectObject:atIndexPath:` of `TTTableViewController`. In this method we create a `TTURLAction` object, supply an instance of `NSDictionary` with custom data into `query` property, and open the URL action by `TTNavigator`. If the `query` property of `TTURLAction` is specified a new view controller is automatically created using `initWithNavigatorURL:query:` method. `URL` and `query` properties are passed into it.

5.1.5 Additional Mapping Features

In map definition we can specify view controllers transition animation type when they are pushed into navigation stack:

```
[map from:@"tt://unitConverter" toViewController:[UnitConverterViewController class] transition:
    UIViewAnimationTransitionFlipFromLeft];
```

Exhibit 8: TTMap specifying view transition

We can set method to be called right after a new view controller has been initialized:

```
[map from:@"tt://categoryIsEmpty" toViewController:[CategoryListViewController class] selector:
    @selector(categoryIsEmptyAlert)]
```

Exhibit 9: TTMap defining selector to be called automatically

We can display new view controller modally:

```
[map from:@"tt://grocery" toModalViewController:[GroceryListViewController class]]
```

Exhibit 10: TTMap to modal view controller

5.1.6 Pushing New View Controllers Using Actions

There are several ways how to push new view controller into the navigation stack. In `TTTableViewController` we can define `dataSource` property with a set of `TTTableItems`. Each `TTTableItem` has the `URL` property. When it is specified to a value, a particular URL is opened by `TTNavigator` when tapping the table item.

We can also use advantage of `NSString` category methods `openUrl` or `openUrlFromButton:`. Method `openUrlFromButton:` is very suitable to be assigned as an action method for custom buttons or navigation bar items.

5.2 Enhanced Table Views

Three20 library extends the basic Cocoa Touch concept of table views, where the main role plays the table view controller with its data source and the delegate including table view with table view cells. Three20 introduces `TTTableItem` classes encapsulating table view cells. To manage table items data source and delegate protocols are extended to `TTTableViewDataSource` and `TTTableViewDelegate` protocols and default classes implementing them are present. We said that the data source can be considered as a kind of model for the table view controller. In Three20 there is a real model – `TTModel` – handling access to physical data. The data source is supposed to observe this model and forward its changes to the table view. `TTTableViewController` knows about the `TTModel` because it extends the `TTModelViewController`. The table view remains an instance of `UITableView` and displays table view cells craftily encapsulated into table items.

5.2.1 Three Layer Concept

Let us simplify the concept into three layers - table view controller, data source plus delegate and model. To get it all work we need to implement several methods across all the layers.

5.2.1.1 Table View Controller

1. `init` method – here we specify title for navigation bar or setup table view properties such as `variableHeightRows` or `tableViewStyle`.

2. `createModel` – in this method we initialize the instance of our data source. If the data source is not using the model object we create here all the table items by ourselves.

3. `createDelegate` – in case we need our customized delegate we specify it in this method. However, we mostly do not need to do that.

5.2.1.2 Data Source

1. `init` – we initialize model object here.
2. `model` – returns the model instance casted to type of `id<TTModel>`.
3. `titleForLoading:`, `titleForEmpty`, `subtitleForError:` – in these methods we specify title for table view when it is loading, empty or when any error occurred.
4. `tableViewDidLoadModel:` – this is the *core* method of data source. Here we ask the model for data it loaded, create an array of table items and assign them to `items` property.

5.2.1.3 Model

1. `isLoaded` method – asks whether model is already loaded, it happens after `load:more:` method has been issued.
2. `isLoading` – asks whether model is currently loading. It can take seconds to load all the data from network. During that time title for loading and a spinner circle are displayed in the table view.
3. `isOutdated` – asks whether model is outdated. After model state changes it does not need to reload itself immediately, it is enough to set model state as outdated and it will be reloaded automatically when needed.
4. `load:more:` – here actual loading goes. In this method we fetch database data or access the internet (loading data from internet is a bit more complicated, we will explain it later)

5.3 Table View Item and Its Customization

Default Three20 table view concept says that table view consists of table items. A table item is basically a row displayed in table view. Three20 offers a really rich set of predefined table items to accomplish almost anything we need. Namely, there are single lined text items, items including a picture within them, items displaying title, caption and text, items with variable length text or items enabled to show text formatted as HTML.

All these items are subclasses of `TTTableItem` class. This class does not provide much functionality itself. It only specifies that table items are serializable using the `NSCoding` protocol. More useful items are those ones subclassing the

`TTTableLinkedItem` class because it defines properties `URL` and `accessoryURL` and thus allows the programmer to perform actions when the item is tapped in the table view. Usually, when a user taps on a table item, `URLAction` with `URL` property is performed by `TTNavigator`. When a property `accessoryURL` is defined, an accessory disclosure button is shown on the left side of the table row and `URLAction` to `accessoryURL` is performed when this button is tapped.

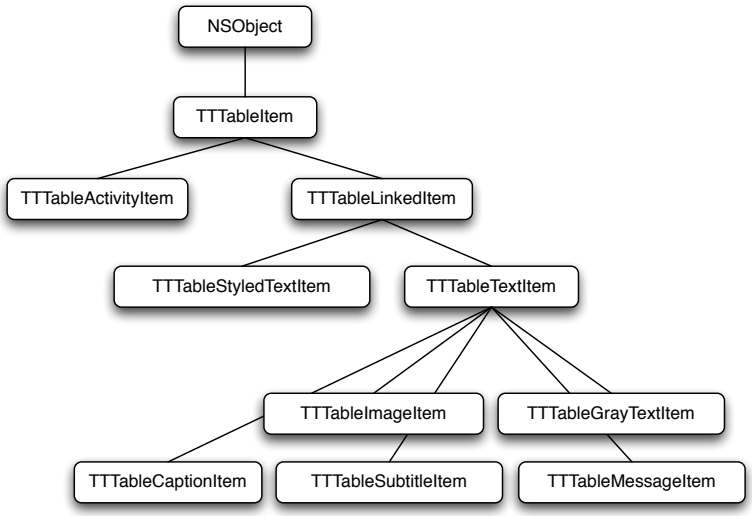

Figure 5: Selection of TTTableItem subclasses hierarchy

`TTTableTextItem` and its subclasses provide convenient methods for easy creation of table items. Methods have signatures similar to:

```
+ (id)itemWithText:(NSString*)text URL:(NSString*)URL accessoryURL:(NSString*)accessoryURL;
```

Exhibit 11: TTTableTextItems convenient methods signature

As we go further in the `TTTableItem` inheritance tree items are becoming more and more complex. Extended with titles, captions, images, links etc. A great overview is provided in the TTCatalog App which comes with sources of Three20 framework available on `http://github.com/facebook/three20`.

5.3.1 Table View Creation Using Table Items

Table items must be assigned to a table view to let it know what to display. We can accomplish this in the class implementing `TTTableViewDataSource` protocol and acting as `dataSource` in `TTTableViewController`. This protocol inherits from `UITableViewDataSource`, `UISearchDelegate` and `TTModel` protocols. The class `TTTableViewDataSource` implementing `TTTableViewDataSource` protocol is already provided in Three20 framework. It handles lots of desirable behavior, but in fact, this class does not allow to insert rows into the table view. For that purpose there is the `TTListDataSource` class which subclasses `TTTableViewDataSource` and has property `items` dedicated to store items in table view.

`TTListDataSource` is designed to operate non sectioned table view, if we need a sectioned one, we can the define data source subclassing `TTSectionedDataSource` class where the `sections` property specifies items sections. For both of them, there are convenient methods for easy insertion of table items.

There are also two more default data sources – `TTThumbsDataSource` intended to be a source of thumbnails pictures and `TTTableViewInterstitialDataSource` which is according to authors definition „A data source that is externally loading. Useful when you are in between data sources and want to show the impression of loading until your actual data source is available."

When we choose a data source class which fulfills our needs we add items into it either in its `init` method or more likely in `tableViewDidLoadModel:` method called after the model has been successfully loaded. When we are displaying only static data we can save time and effort and create the data source manually, using some of the convenient methods and assign it to `dataSource` property in `createModel` method of `TTTableViewController`.

5.3.2 Defining Custom Table Items

When defining custom table items we need to meet basic principles of table items functionality.

As we know, `TTTableView` displays `UITableView` and `UITableView` requires `UITableViewCells` as its rows. Because of that, each `TTTableItem` subclass also needs to specify its `TTTableViewCell` subclass to link each item with the corresponding table view cell. Thus, for each table item there is a corresponding table view cell. For instance, `TTTableCaptionItem` works with `TTTableCaptionItemCell` and so forth.

There are three actors playing the role when defining our own table items:
`TTTableItem`, `TTTableViewCell`, and `TTTableViewDataSource`. General guideline of creating custom table item is:

1. to define subclass of `TTTableItem`
2. to define subclass of `TTTableViewCell`
3. to assign certain table view cell to certain table view item.

In our sample App we defined custom table item for displaying the list of categories. Here is how we assigned our `CategoryTableItem` class to use `CategoryTableViewCell` class:

```
// in CategoryListDataSource
- (Class)tableView:(UITableView*)tableView cellClassForObject:(id) object {

    if ([object isKindOfClass:[CategoryTableItem class]]) {
        return [CategoryTableViewCell class];
    } else {
        return [super tableView:tableView cellClassForObject:object];
    }
}
```

Exhibit 12: Assigning table item cell to table item

The objective of table items is to be serializable using `NSCoding` protocol, so basically we only need to implement some convenient methods and the `NSCoding` protocol methods in our custom table item.

Then implementing of the table view cell goes on turn. Here we should definitely implement methods `initWithStyle:reuseIdentifier:` and `object` property accessors where we get and set the table item. Very probably we will also need to implement `tableView:rowHeightForItem:` convenient method and `layoutSubviews` method to layout elements in the table view cell properly.

When all of this is done we make sure not to forget to override `tableView:cellClassForObject:` method of `TTTableViewDataSource` to link our custom table item to the table view cell.

Now we can assign our custom table items to the data source in method `tableViewDidLoadModel:` or alike and display them in a table view.

5.4 Networking the Easy Way

Three20 uses concept of models and their view controllers. Networking here is very easy because there is already pre-built model class `TTURLRequestModel` which handles most of the necessary work for us. In addition, Three20 networking classes offer functionalities such as caching network requests into the disk (normally only RAM caching is available), running network communication in separate threads with build-in locking and concurrency management or suspending and resuming loading requests to improve application performance.

5.4.1 Getting Data From the Web

When our App needs to get any data from the internet it is as easy as implementing two methods of `TTURLRequestModel`.

- First, we implement `load:more:` method, where we create `TTURLRequest` object with data we need to send to specific URL destination
- Second, we process incoming data from the request in `requestDidFinishLoad:` method. The data loaded from the network we get in methods argument. We always make sure to call `[super requestDidFinishLoad:]` at the end of the method.

When using `TTURLRequestModel` we do not take care about whether model `isLoading`, `isLoaded` or `isOutdated`. The class itself will handle it for us.

5.4.2 Setting up URL Request Parameters

The `TTURLRequest` class has several properties to specify network request such as destination URL, GET and POST data or more advanced settings such as loading policy type or cache expiration age.

Here is how can we create URL request sending data to specific URL in our sample App:

```
- (void)load:(TTURLRequestCachePolicy)cachePolicy more:(BOOL)more {
    if (! self .isLoading) {
        _isOutdated = YES;
        NSString* url = [NSString stringWithFormat:[self sourceUrlFormat], [self
            sourceUrlFormatVariable]];

        TTURLRequest* request = [TTURLRequest requestWithURL:url delegate:self];

        id<TTURLResponse> response = [[TTURLDataResponse alloc] init];
        request.response = response;
        TT_RELEASE_SAFELY(response);

        request.cachePolicy = cachePolicy;
        request.cacheExpirationAge = TT_CACHE_EXPIRATION_AGE_NEVER;

        [request send];
    }
}
```

Exhibit 13: Creating and sending URL request

5.5 Simulating iPhone Springboard

The Springboard is the first screen we see when unlocking the iPhone device. It shows all the applications and launches them on finger tap. In Three20 framework there is the `TTLauncherView` class and the `TTLauncherViewDelegate` protocol simulating Springboard behavior and allowing us to run *sub-applications* within our App. It should be, however, clearly recognizable that the sub-application has been launched from a launcher view, otherwise user may think he/she is running a normal application from Springboard and may want to close it by pressing home button. Which would obviously close the whole App not only the sub-App from the launcher view.

Launcher view offers many features like rearranging of sub-Apps icons, deleting or adding new ones, specifying number of launcher pages and rows in a page etc.

5.5.1 Embedding Sub-Apps Into App

General approach to embed a launcher view into an App is to:

1. create `TTViewController` with property of `TTLauncherView`
2. assign class conforming to `TTLauncherViewDelegate` protocol to `delegate` property of `TTLauncherView` class. Usually, `TTViewController` subclass also adopts `TTLauncherViewDelegate` protocol thus the property is set to `self`.
3. create and assign `TTLauncherItem`s to launcher view and do other setup in view controllers `didLoad` method.

In the sample App we use a launcher view to run the unit converter and the timer:

```
- (void)loadView {
    [super loadView];

    _launcherView = [[TTLauncherView alloc] initWithFrame:self.view.bounds];
    _launcherView.delegate = self;
    _launcherView.columnCount = 4;
    _launcherView.pages = [NSArray arrayWithObjects:
        [NSArray arrayWithObjects:
            [[[ TTLauncherItem alloc] initWithTitle :@"Unit␣convert"
            image:@"bundle://convert.png"
            URL:@"tt://unitConverter" canDelete:NO] autorelease],
            [[[ TTLauncherItem alloc] initWithTitle :@"Timer"
                image:@"bundle://timer.png"
                URL:@"tt://timer" canDelete:NO] autorelease],
            nil ],
        nil ];

    [ self .view addSubview:_launcherView];
}
```

Exhibit 14: Example of TTLauncherView initialization

When a launcher item is selected we want to push new view controller. We can do it very easily using URL for `TTNavigator`, thus letting animation and behavior of launching be defined in `TTURLMap`:

```
- (void)launcherView:(TTLauncherView*)launcher didSelectItem:(TTLauncherItem*)item {
    [item.URL openURL];
}
```

Exhibit 15: Handling launcher item click

5.6 CSS-like Stylesheets

The `TTStyle` in Three20 helps to specify items appearance in runtime. There is no need to hard code it in `drawRect:` method of `UIView`. Styles can be assigned and replaced in runtime.

There is one global stylesheet class – `TTDefaultStyleSheet` – defining appearance for whole application. It defines colors of toolbar or navigation bar, background color for table views, font types for labels and many others. If we want to change those things we may subclass `TTDefaultStyleSheet` class and redefine corresponding methods. Redefining one method will cause whole application to use this definition.

For instance we can change background color for all views and for navigation bar by overriding two methods:

```
- (UIColor*)backgroundColor {
    return RGBCOLOR(209, 246, 212);
}
- (UIColor*)navigationBarTintColor {
```

```
        return RGBCOLOR(0, 140, 0);
    }
```

Exhibit 16: Redefining of TTDefaultStylesheet properties

Styles in Three20, however, offer much more functionality. There are approximately 40 classes related to `TTStyle` which we can use to define custom shapes, labels with backgrounds, buttons etc. Some *TT* components directly supports `TTStyle`s for it's custom rendering in runtime. Examples are:

1. `TTButton`
2. `TTTabBar`
3. `TTSearchBar`
4. `TTStyledFrame`

5.6.1 Custom Buttons Styling

As an example, we created an instance of `TTButton` class and specified its appearance by `TTStyle`. The button can change its appearance in runtime by choosing another `TTStyle` to work with. In the sample App we defined two style methods:
`timerStartButton:` and `timerStopButton:`. Then we initialized the button to use one of the methods. We are able to change this style anytime in runtime as shown in the following exhibit:

```
        // init button with style timerStartButton
        _button = [[ TTButton buttonWithStyle:@"timerStartButton:" title :@"Start"] retain ];
        _button.frame = CGRectMake(20, 300, 280, 40);

        //change button style to timerStopButton
        [_button setStylesWithSelector:@"timerStopButton:"];
```

Exhibit 17: Assigning style property to TTButton

6 Communicating to RESTful Web Services

RESTful services allow to access data at specified URL. The data are accessed using HTTP protocol methods such as GET, POST, PUT and DELETE. Most RESTful services, however, use the POST method instead of PUT and DELETE. Unlike procedure oriented SOAP protocol, REST is oriented to data. Each data resource is accessible on its own URL.

RESTful interfaces generally have several advantages over SOAP protocol. They are:

- lightweight – not a lot of unnecessary XML markup
- human readable – result are usually easy to read, thus, easy to process
- no tools required – structure of REST message is simple, no complex parsing libraries are needed

Because of iPhone network access limitations (many iPhone devices still communicate over EDGE or GPRS, which is quite slow) we would prefer to use RESTful services over the SOAP to access data on remote servers. Firstly, because the SOAP protocol is too verbose which causes needless network traffic. And Secondly, there are no libraries encapsulating SOAP communication. Only `NSURLRequest` and XML parsing framework are available, which is pretty little to implement SOAP communication, unless we want to spent tons of hours getting it to work.

In the sample App we chose JSON protocol for REST resource representation. For parsing JSON strings we used open source SBJSON library hosted on Google Code (`http://code.google.com/p/json-framework/`).

6.1 SBJSON Framework

SBJSON framework allows to parse JSON data very easily. Each string containing JSON representation of an object is parsed into `NSDictionary` object, the `string`s and `value`s are transformed into keys and values in the dictionary. JSON representation of an array is transformed into `NSArray` object usually containing dictionaries as its entries.

White characters such as tabulator (`\t`) or new line (`\n`) character are properly handled as well.

6.1.1 Getting Objects from REST Resources

There are several approaches how we can utilize SBJSON framework capabilities.

1. The `SBJsonParser` and `SBJsonWriter` classes allow us to encode an object into JSON string or decode JSON string by hand. We have full control over parsing of our data using this approach but it requires more code to be written. Thus bigger chance of a mistake.

2. Better way is to use advantages of `SBJSON` class acting as a facade simplifying `SBJsonParser`s and `SBJsonWriter`s interface. It comes very handy when we want to track errors while parsing the data. Each method gets a `NSError` argument which, in case of any error, is filled in by the error description. As an example we can use `objectWithString:error:` method to decode JSON string to `NSArray` or `NSDictionary` respectively or `stringWithObject:error:` method to encode given object to its JSON representation.

3. If we do not need to track errors, the simplest way is to use `NSString` and `NSObject` category additions which allow to encode and decode objects even easier. We can use `JSONRepresentation` method to get an objects JSON representation or `JSONValue` method to encode an object into JSON string.

When we want to transfer REST resource from remote server into our custom object we subclass `TTURLRequestModel` and in the `requestDidFinishLoad:` method we parse received data and fill in a property containing resulting objects. Given example is slightly modified source code for parsing recipe data from the sample App.

```
- (void)requestDidFinishLoad:(TTURLRequest*)request {

    TTURLDataResponse *response = request.response;
    // This response is designed for NSData objects, so if we get anything else it's probably a
    // mistake.
    TTDASSERT([response.data isKindOfClass:[NSData class]]);

    if (_jsonString != nil)
        TT_RELEASE_SAFELY(_jsonString);

    //set ivar to received http response body
    if ([response.data isKindOfClass:[NSData class]]) {
        NSString *recipeDataStr = [[NSString alloc] initWithData:response.data encoding:
            NSUTF8StringEncoding];
        _jsonString = recipeDataStr;
    }

    NSArray *recipesDicts = [_jsonString JSONValue];

    TT_RELEASE_SAFELY(_recipes);
    NSMutableArray *recipes = [[NSMutableArray alloc] init];

    for (NSDictionary *singleRecipeDict in recipesDicts) {

        [recipes addObject:[[[Recipe alloc] initWithDictionary:singleRecipeDict] autorelease]];
    }
    _recipes = recipes;

    [super requestDidFinishLoad:request];
    _isOutdated = NO;
}
```

Exhibit 18: Example of parsing JSON string from URL response

7 Conclusion

The aim of the bachelor thesis was to explain basics of iPhone applications development and demonstrate them on sample application.

We tried to provide background for iPhone programming beginners. We explained fundamentals of Cocoa Touch framework which is the minimum programmer must know and then we moved to more advanced topics and described external open source frameworks such as Three20 framework and SBJSON library. To avoid information overload we excluded section about Objective-C language syntax. Reader who is not skilled in it needs to learn it by him/herself. The Apple Documentation, however, provides many resources in this field as well.

The sample application is ready to be put into the AppStore but it would have not many chances to success since its functionality is already fulfilled by bunch of other Apps doing similar task. Moreover, the application graphics is not good enough to beat its AppStore competitors. To be really prepared for launching it we would have to add more features to it, improve application graphics and prepare resources for marketing before the App can be introduced to the rest of the word.

8 References

[1] Christopher Allen, Shannon Appelcline. *Iphone in Action: Introduction to Web and SDK Development*, Manning Publications Co., 2009, ISBN 193398886X.

[2] Erica Sadun. *The iPhone Developer's Cookbook, Building Applications with the iPhone SDK*, Addison-Wesley, 2009, SBN-10: 0-321-55545-7.

[3] Jonathan Zdziarski. *iPhone SDK Application Development, 1st Edition*, O'Reilly Media, Inc., 2009, ISBN-13: 978-0-596-15405-9.

[4] Apple Developer *iPhone OS Reference Library*, `http://developer.apple.com/iphone/library/` Apple Inc., 2010.

[5] Alan Cannistraro, Josh Shaffeer. *iPhone Applicaiton Development (Winter 2010)* Stanford University on iTunes U, 2010.

[6] Jeff Verkoeyen. *Three20 framework official webpage*, `http://three20.info/`

[7] How to Make iPhone Apps. *iPhone OS Design Patterns*, `http://howtomakeiphoneapps.com/2009/06/iphone-os-design-patterns/`

[8] Cocoa Dev Central. *Core Data Class Overview*, `http://cocoadevcentral.com/articles/000086.php`

[9] Sen:te. *Sen:te - OCUnit*, `http://www.sente.ch/software/ocunit/`

[10] Wikipedia. *Waterfall model*, `http://en.wikipedia.org/wiki/Waterfall_model`

[11] Wikipedia. *Agile software development*, `http://en.wikipedia.org/wiki/Agile_software_development`

[12] Mat's Portfolio. *Three20 TTTableItem Tutorial*, `http://mattvague.com/three20-tttableitem-tutorial`

[13] Google Groups. *Three20 group on Google Groups*, `http://groups.google.com/group/three20/`

[14] Wikipedia. *Representational State Transfer*, `http://en.wikipedia.org/wiki/Representational_State_Transfer`

[15] Pete Freitag. *REST vs SOAP Web Services*, `http://www.petefreitag.com/item/431.cfm`

[16] JSON.org. *JSON official webpage*, `http://json.org/`

A Useful Links

URL	What information is it source of?
http://github.com/facebook/three20	Three20 framework source files on github
http://three20.info/	Official Three20 framework webpage containing instalation guide, API reference and few tutorials
http://developer.apple.com/iphone/	iPhone Dev Center containing tons of documentation and examples
http://groups.google.com/group/three20/	Three20 framework group on Google Groups
http://code.google.com/p/json-framework/	SBJSON framework source code on Google Code
http://itunes.stanford.edu/	iPhone Application Development lectures by Stanford University on iTunes U
http://cvut.iknow.eu/iphone/	Czech iPhone programming lectures at ČVUT University
http://iPhoneDeveloperTips.com/	Quick tutorials to various topics in iPhone and iPad development
http://icodeblog.com/	Another source of useful tutorials
http://iphonedevelopmentbits.com/	A blog collecting plenty of interesting articles related to iPhone development
http://www.cimgf.com/	Articles about more experienced topics regarding iPhone programming

B Content of Enclosed CD

iPhoneApp – source codes of sample iPhone application

webApp – source codes of sample web application

thesis – this text